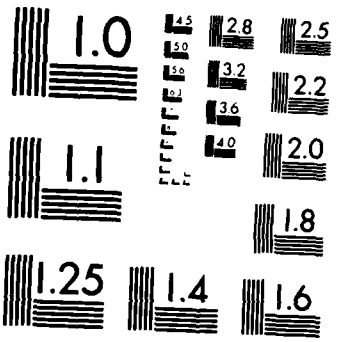
AD-A147 213 SUPPORT FOR DISTRIBUTED TRANSACTIONS IN THE TABS (TRANSACTION BASED SYSTE..(U) CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE 1/1

UNCLASSIFIED A Z SPECTOR ET AL. JUL 84 CMU-CS-84-132 F/G 9/2 NL



CMU-CS-84-132

12

Support for Distributed Transactions in the TABS Prototype

Alfred Z. Spector, Jacob Butcher, Dean S. Daniels,
Daniel J. Duchamp, Jeffrey L. Eppinger, Charles E. Fineman,
Abdelsalam Heddaya, Peter M. Schwarz

July 1984

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

REPORT NUMBER: CMU-CS-84-132	GOVT ACCESSION NO: AD A147 213
RECIPIENT'S CATALOG NUMBER:	
TITLE (and Subtitle): Report for Distributed Transactions in the TABS Prototype	TYPE OF REPORT & PERIOD COVERED: Technical Report
	PERFORMING ORG. REPORT NUMBER:
AUTHOR(s): A. Spector, J. Butcher, D. Daniels, D. Duchamp, J. Eppinger, C. Fineman, A. Heddaya, and P. Schwarz	CONTRACT OR GRANT NUMBER(s): N00014-79-C-0661
PERFORMING ORGANIZATION NAME AND ADDRESS: Carnegie-Mellon University, Department of Computer Science, Schenley Park, Pittsburgh, PA 15213	PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61153N 14, RR01408, RR0140801, NR 049-447
CONTROLLING OFFICE NAME AND ADDRESS: Office of Naval Research (Code 433), 800 N. Quincy St., Arlington, VA 22217-5000	REPORT DATE: July 1984
	NUMBER OF PAGES: 25
MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	SECURITY CLASS. (of this report): unclassified
	DECLASSIFICATION/DOWNGRADING SCHEDULE

DISTRIBUTION STATEMENT (of this Report)

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

SUPPLEMENTARY NOTES

KEY WORDS (Continue on reverse side if necessary and identify by block number)

ABSTRACT (Continue on reverse side if necessary and identify by block number)

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Support for Distributed Transactions in the TABS Prototype

Alfred Z. Spector, Jacob Butcher, Dean S. Daniels,
Daniel J. Duchamp, Jeffrey L. Eppinger, Charles E. Fineman,
Abdelsalam Heddaya, Peter M. Schwarz

July 1984

Abstract

The TABS Prototype is an experimental facility that provides operating system-level support for distributed transactions that operate on shared abstract types. It is hoped that the facility will simplify the construction of highly available and reliable distributed applications. This paper describes the TABS system model, the TABS prototype's structure, and certain aspects of its operation. The paper concludes with a discussion of the status of the project and a preliminary evaluation.

Technical Report CMU-CS-84-132

This work was supported by the IBM Corporation, the National Science Foundation, and the Defense Advanced Research Projects Agency, ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of any of the sponsoring agencies or the US government.

Table of Contents

A

A1

1. Introduction

The TABS (Transaction Based Systems) Prototype is an experimental facility that provides operating system-level support for distributed transactions that operate on shared abstract types. We are building the TABS prototype both to gain experience in implementing general purpose transaction mechanisms and to provide additional evidence that the transaction concept simplifies the construction of highly available and reliable distributed applications [Spector 83]. Others also believe that transactions are a useful programming construct, and some groups are also building systems with goals that are similar to our own [Gray 81a, Liskov 82, Liskov 83, Allchin 83a].

TABS supports transactions containing operations on objects that are instances of user-defined, abstract data types that do type-specific synchronization and recovery. Recent papers by ourselves and others have identified type-specific aspects of synchronization and recovery that can be used by these abstract data types [Weihl 83, Allchin 83b, Daniels 83, Schwarz 83, Schwarz 84a, Korth 83]. In TABS, the responsibility for recovery and synchronization is divided between the system and individual types in a fashion that contributes to efficient operation and provides for composition of operations on different objects.

Transactions in TABS can have their usual properties: failure atomicity, serializability, permanence, and freedom from cascading aborts. Failure atomicity ensures that a transaction's partial results are undone when it is interrupted by a failure. Programmers are therefore free to violate consistency constraints temporarily during the execution of a transaction. Serializability ensures that other concurrently executing transactions cannot observe these inconsistencies. Permanence ensures that modifications performed by committed transactions are not lost due to node or media failures. Prevention of cascading aborts limits the amount of effort required to recover from a failure. Together or individually, these properties can simplify the treatment of failures and concurrency in distributed programs. See Gray for a brief tutorial on transactions [Gray 80].

TABS does not require that all transactions have these properties associated with them, even though using transactions without all four properties may increase the complexity of writing TABS programs. In some instances, the increased performance that results may outweigh this complexity. For example, to gain increased concurrency, transactions are permitted to use objects that do not preserve the serializability of transactions that use them.

TABS provides transaction support at a relatively low level. In addition to permitting more efficient implementations, having a uniform transaction facility at a low level should simplify the composition of activities that are implemented in different subsystems, such as a relational database system and a

hierarchical file system. Eventually, it may be desirable to integrate transaction support into the lowest levels of the operating system or possibly to provide special hardware facilities to make transactions execute more efficiently.

Though still evolving, the TABS prototype has experimentally executed its first transactions[1]. The implementation is on a modified version of the Accent Kernel [Rashid 81] running on Perq 2 computers that are interconnected by a 10 Mbit/sec Ethernet. In many ways, the TABS Prototype will have poor performance, particularly because of its reliance on message communication between processes with separate address spaces. However, many aspects of its design are sensible and, we believe, worthy of attention.

After introducing the TABS transaction and object models in the next section, Section 3 presents an overview of the major components of the TABS system. Section 4 describes how the components interact during system operation. Section 5 concludes with a discussion of the status of the system, our plans for experiments, and our initial evaluations of its design.

2. The TABS System Model

The two primary components of the TABS system model are transactions and objects. Processes may initiate transactions and then invoke operations on any number or type of objects, located locally or remotely. If all of the operations in the transaction complete successfully, the initiating process may request that the transaction be committed. Prior to transaction commit, any of the processes involved in the transaction may request that the transaction be aborted.

Objects in TABS are instances of abstract data types and are encapsulated in processes called *data servers*. An operation on an object is invoked via a request message to the data server containing it. When it has finished executing the operation, the data server then sends a response message containing the result. In TABS, a requesting process may initiate multiple operations that proceed in parallel. In addition to executing operations, data servers contribute to the synchronization, recovery, and committing and aborting of transactions. In many ways, data servers correspond to the Guardians provided by the Argus programming language [Liskov 82].

To implement operations, data servers may modify and read data that they store within them, and they may invoke operations on other data servers. Data servers support the synchronization

---

[1] This paper reflects the state of the current system, as well as code that is still being implemented. Section 5.1 describes the project's status in more detail.

requirements of transactions by using locking to synchronize access to the objects they read and modify [Gray 80]. However, rather than setting only Read and Write locks on objects, data servers can exploit the semantics of operations to achieve increased concurrency using type-specific locking [Schwarz 84a, Korth 83]. In type-specific locking, every operation on an abstract object acquires a lock from the set of lock modes associated with that object. A type-specific lock compatibility relation is used to determine whether a lock may be acquired by a particular transaction.

With the aid of other TABS components, data servers also support the failure atomicity and permanence requirements of objects. TABS uses recovery techniques based upon write-ahead logging, rather than shadow pages [Gray 78, Gray 81b, Lindsay 79, Schwarz 83, Schwarz 84b, Lampson 81, Lorie 77]. The advantages of logging have been discussed elsewhere and include the potential for higher concurrency, reduced I/O activity at transaction commit time, and the potential for contiguous allocation of objects on secondary storage [Gray 81b, Traiger 82, Schwarz 83]. For flexibility, two separate write-ahead log algorithms are implemented.

The simpler of the two TABS recovery algorithms is based on value logging. In this approach, data servers log the old and new values of objects, or portions thereof, whenever they update failure atomic or permanent objects. During recovery processing, the portions of a data server's virtual memory containing an object are reset to the object's most recently committed values during a one pass scan that begins at the last log record written and proceeds backward. If the value logging algorithm is used, only one transaction at a time may modify an object.

In the operation-based recovery algorithm, data servers write log records containing the names of operations and enough information to undo them. Operations are redone or undone, as necessary, during recovery processing to restore the correct state of objects. An important feature of this algorithm is that operations on multi-page objects can be recorded in one log record. The operation-based recovery algorithm also permits a greater degree of concurrency than the value based recovery algorithm and may require less log space. However, it is more complex, and it requires three passes over the log during crash recovery, instead of the single pass needed for the value based algorithm.

The TABS recovery algorithms are flexible enough to be used for many varieties of shared abstract types. Unfortunately, the algorithms are complex, and require lengthy descriptions. However, the reader who is familiar with previously published write-ahead log-based algorithms [Gray 78, Lindsay 79] possesses a basic understanding of how the TABS algorithms work. The details are fully described in other reports [Schwarz 83, Schwarz 84b].

From the perspective of control flow, a transaction begins in one process, but may continue execution in many others. In fact, TABS permits transactions to execute in parallel on multiple data servers; this is useful, for example, to support distributed replication algorithms. From the point of view of a data server, there may be many uncommitted transactions active at any given time. Because of this, all messages in TABS contain not only a destination, but also a unique *transaction identifier* that specifies the transaction on behalf of which the message was sent.

To reduce the concurrency anomalies that can occur within a single transaction that has simultaneous threads of control, and to permit portions of a transaction to abort independently, TABS also supports a limited subtransaction facility. This facility can be largely characterized by its synchronization and commit policies:

- A subtransaction behaves as a completely separate transaction with respect to synchronization. This provides protection between simultaneous threads of control.

- A subtransaction is not committed until its top level parent transaction commits, but a subtransaction can abort without causing its parent transaction to abort. Permitting subtransactions to abort independently is useful, for example, if transactions can tolerate the failure of some of the operations that they initiate.

The TABS subtransaction model is simpler than the nested transaction models of Reed [Reed 78] or Moss [Moss 81], though it is certainly less flexible. We hope that experimentation with the TABS prototype will tell us if it is powerful enough to program data servers that use nested abstractions.

3. The Structure of TABS

TABS has seven major components, six of which are realized by one or more Accent processes; the other is the Accent Kernel itself. The two user-programmable components of TABS are *data servers*, which have been mentioned previously, and *application processes*, which initiate transactions. (Also, see Figure 3-1.) All the components of TABS are briefly described below:

- The *Accent Kernel* supports processes with protected, virtual address spaces that communicate via message passing.

- *Data Servers* store long-lived data in their virtual memory and use type-specific locking for synchronization. Data servers, the Accent Kernel, and the Recovery Manager cooperate to implement a write-ahead log-based recovery mechanism.

- *Name Server* processes on each node cooperate to map external object names to particular objects within data servers.

- *Application processes* initiate transactions and invoke operations in data servers.

- The *Recovery Manager* has three major functions. During normal operation, the Recovery Manager receives records from data servers and the Transaction Manager and writes these records to the log. The Recovery Manager presides over transaction abort and recovery after node crashes.

- The *Transaction Manager* coordinates the initiating, committing and aborting of local and distributed transactions.

- The *Communication Manager* relays inter-node messages and also records information that is used for crash detection and transaction commit.

Sections 3.1 to 3.7 describe the TABS components in more detail.

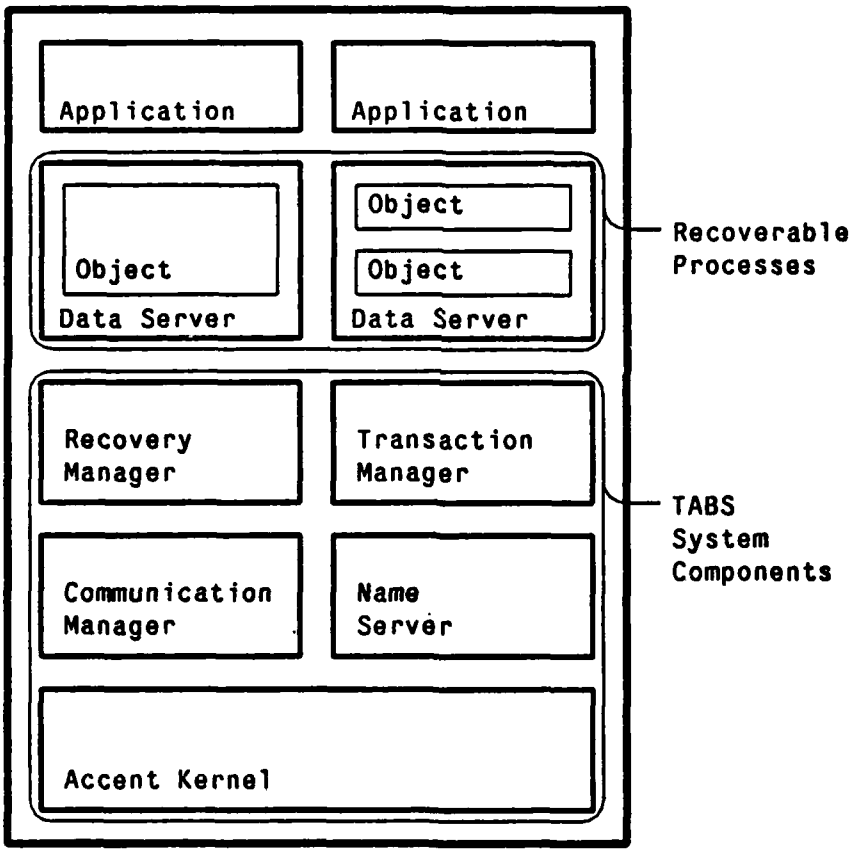


Figure 3-1: The Basic Components of a TABS Node

3.1. The Accent Kernel

The Accent Kernel provides many of the underlying facilities needed to implement a transaction mechanism including primitives for creating, destroying, and scheduling processes [Rashid 81, Rashid 83]. Each Accent process has a private 2^{32} byte virtual address space, which can be sparsely allocated. Accent supplies several kinds of memory objects, called *segments*, that can be used to store both temporary and long-lived data. Processes communicate by sending messages to a *port* where they are queued for the recipient process. The same mechanism is also used for communication between processes and the Accent Kernel. Accent guarantees that intra-node messages are delivered at most once and are not reordered. With the cooperation of the Communication Managers, inter-node communication is location-transparent.

The standard Accent Kernel can provide a process with access to a *permanent segment*, which is a region of non-volatile storage that can be mapped into the virtual address space of a process. However, there are two reasons why permanent segments are inappropriate for storing TABS objects. The first is that changes made to the volatile representation of a permanent segment are not written through to the corresponding region of non-volatile storage until an explicit kernel call is made. This is too slow for the TABS log-based recovery techniques that permit updates in place. The second reason is that Accent provides no facilities for interacting with the paging manager that controls permanent segments. Write-ahead logging requires such interaction for data servers so they can temporarily lock pages in main memory, and for the Recovery Manager so logging activity will reflect the movement of pages between main memory and non-volatile storage.

For these reasons, a new type of memory object, the *recoverable segment*, has been added to Accent. This type of segment has been strongly influenced by Traiger's discussion of virtual memory management for database systems [Traiger 82]. Recoverable segments are similar to permanent segments in that they are backed by non-volatile storage. However, the backing storage for a recoverable segment is permanently assigned, and the paging system updates this storage directly.

To support the write-ahead log algorithms used by TABS, each recoverable segment has a port associated with it. The kernel uses this port to send three separate types of messages to the Recovery Manager. The first message indicates that an in-memory page of a recoverable segment has been modified for the first time since it has been paged in. To detect when this message should be sent, the kernel traps the first write reference. The second message indicates that the kernel wants to copy a modified page of the recoverable segment to non-volatile storage. The kernel does not write the page until it receives a message from the recovery manager indicating that all log records that apply to this page have been written to non-volatile storage. The third and final message indicates that a

page in a recoverable segment has been successfully copied to non-volatile memory.

In addition to the special messages that support the write-ahead log algorithms, the Accent Kernel also supplies the **Pin** and **UnPin** primitives for pages of recoverable segments. These primitives are invoked by data servers to prevent Accent from copying pages to disk while operations on objects they contain are in progress.

A final modification to Accent has been made to support the TABS operation logging recovery algorithm. This algorithm requires that every time the kernel copies a page of a recoverable segment to non-volatile storage, it must atomically write a sequence number along with the page data. This sequence number is stored in header space that is available on the Perq disk. The Recovery Manager sends the sequence number to the kernel in the message that indicates that the page can be written to disk. When the Recovery Manager wishes to read a page's sequence number, it sends a request message to the kernel.

3.2. Recovery Manager

The TABS Recovery Manager coordinates access to the log, an append-only sequence of records on non-volatile storage[2]. A node's log contains enough information to restore the state of objects in the node's recoverable segments and to restore the state that the Transaction Manager and data servers require for the two-phase commit protocol. During normal operation, the Recovery Manager receives log data in messages from other TABS components and writes it in the log. During transaction abort and crash recovery, the Recovery Manager reads the log and sends log data to other TABS components. Finally, the Recovery Manager coordinates system *checkpoints*; checkpoints limit the amount of log data that must must be available for crash recovery and reduce the amount of time to recover after a crash.

The Recovery Manager writes log records in response to messages sent by data servers, the Transaction Manager, and the Accent Kernel. The latter messages are sent when pages are first modified and when pages have been written to non-volatile storage. All log records except some that are written by the Transaction Manager are written to non-volatile storage only when a volatile buffer fills or when needed to satisfy the write-ahead log protocol requirements. In particular, the Recovery Manager forces any log records stored in volatile storage that correspond to a page that Accent plans to write out. Log records are always written in the order that the associated messages are received by

---

[2]The log should be on *stable storage*; however, because of our Perq hardware restrictions (only one disk and controller), the non-volatile storage used for the log is *not* stable storage. The eventual implementation of stable log storage in TABS will depend upon the storage technology available.

the Recovery Manager.

In addition to spooling and forcing data to the log, the Recovery Manager oversees transaction abort. It reads the aborted transaction's log records that describe the operations performed and sends records back to the data servers in reverse of the order they were written. Data servers restore objects according to these records and reply to the Recovery Manager as they process the records. The Recovery Manager logs data when it receives these replies so that the log will reflect the restored or partially restored state of objects, should a crash occur. Of course, the write ahead log protocol continues to be observed during transaction abort.

After a node or media failure, the Recovery Manager scans the log one or more times. Based on the information in the log, the Recovery Manager can determine for each log record written by data servers whether the record needs to be forwarded back to the data server and whether the data server needs to redo or undo the operation. In this way the Recovery Manager assures that objects in recoverable segments reflect only the operations of committed and prepared transactions. Also the Recovery Manager returns enough information to the Transaction Manager for it to reconstruct critical elements of its state after a node crash.

Crash recovery always must always read the portion of the log written after the most recent checkpoint. Depending on the contents of the checkpoint record, earlier sections of the log may also be read, but the most recent checkpoint record contains enough information to determine when crash recovery is complete. Section 4.2.2 describes the information contained in a checkpoint record and the protocol for taking checkpoints.

3.3. Data Servers

Data servers encapsulate two kinds of data: objects and associated synchronization information, and three kinds of code: code implementing operations, code implementing type-specific recovery, and code implementing type-specific synchronization. Each data server accepts messages requesting that operations be performed, and returns results and exceptions in reply messages. Data servers are programmed with the aid of a subroutine library that contains utility code for doing synchronization, for manipulating the coroutines in which operations execute, for performing the data server's role in the prepare phase of two-phase commit, and so forth.

For reasons of efficiency, data servers may encapsulate many objects and these may be of more than one type. Thus, each data server represents one or more abstractions that are to be made available to other data servers or applications. A data server stores some or all of the objects that

constitute the implementation of these abstractions, and uses other data servers to store the rest.

Shared objects that must survive node failures are stored in *recoverable segments*. Data servers must **Pin** and **UnPin** objects or portions of objects while modifying them. Other than this, data servers are not concerned with the movement of objects between volatile and non-volatile memory. After a node failure, new data server processes are started that have access to the same recoverable segments that were used by their predecessors.

The states of the objects stored in recoverable segments are restored after transaction abort and node failure using the TABS recovery algorithms. As mentioned previously, data servers contribute to recovery by providing routines for restoring objects to logged values in the case of value logging, or the undo and redo routines for operation logging. In either case, the recovery code is part of every data server that stores objects of a particular type.

In the TABS implementation of type specific locking, data servers contain the lock compatibility relations, the tables describing which locks are held by which transactions, and the code for manipulating these data structures. The system-wide portion of the synchronization mechanism is contained in the Transaction Manager, which is responsible for notifying data servers of events that affect the status of locks, *e.g.*, transaction abort or commit.

3.4. Name Server

The abstractions represented by data servers are permanent entities that must persist despite node failures, even though the processes (and their corresponding ports) that implement them disappear. For this reason, TABS has a naming service that provides permanent names for objects. The Name Server processes that execute on each node of a TABS network cooperate to perform this service.

Whenever started, each data server registers the global names of the objects it stores with its local Name Server. More specifically, a data server registers the global name of each object with a pair containing its receive port and a *logical object identifier*. The latter distinguishes between the multiple objects that are implemented by a data server. TABS permits multiple data servers to register the same global name with separate pairs of ports and logical object identifiers. This is useful if an object is replicated on multiple data servers. In response to lookup operations for an object on a remote node, the port returned by the Name Server is to the local Communication Manager, which then relays messages across the network to the remote node.

3.5. Applications

Application processes contain the top-level programs that initiate transactions. They are similar to data servers in that they invoke operations on objects by sending messages and accepting replies. However, applications processes store no data themselves and therefore do not accept messages from other data servers. Applications are not affected by node failure recovery, and do not necessarily have to be restarted after a node failure. Of course, any transactions that were in-progress at the time of the failure will be aborted by other TABS components during recovery.

3.6. Transaction Manager

The Transaction Manager is a single Accent process whose primary function is to coordinate the initiation and termination of transactions in conjunction with Transaction Managers at other nodes. Accordingly, each Transaction Manager must at all times be aware of the state of every transaction active at its node. This state information is provided by two kinds of messages that other TABS processes send to it. Some messages request the initiation or termination of transactions; other messages inform the Transaction Manager of what remote sites and local servers are participating in a given transaction.

Messages may be sent by application processes or data servers to begin a transaction, to attempt to commit a transaction, or to force a transaction to be aborted. However, the process that tries to commit a transaction must be the same process that earlier began it. To commit or abort a transaction, the Transaction Managers use a variant of the two-phase commit protocol described by Lindsay, Gray, and Mohan [Lindsay 79, Gray 78, Mohan 83]. The protocol is tree-structured; each node serves as coordinator for the nodes that are its children. The tree used in our algorithm is a spanning tree where a node, A, is a parent of another node, B, if and only if node A was the first node to invoke an operation on behalf of the transaction on node B. The information about a node's relation to the nodes directly above and below it in the spanning tree is kept by its Communication Manager.

There are two messages that TABS components send to inform the Transaction Manager of the progress of a transaction. The first is sent by a data server the first time it takes action on behalf of a particular transaction; doing so informs the Transaction Manager of which servers it must communicate with when the transaction is being terminated. The other message is sent by the Communication Manager the first time that a local process sends a message to a remote server. This message indicates that there are remote sites that have servers active on behalf of the given transaction. At this point, the Transaction Manager becomes aware that remote sites are involved in the transaction, but it cannot identify these sites. This more complete information is obtained from

the Communication Manager during commit processing.

The existence of subtransactions in the TABS model complicates transaction management very little. The same messages that are used to inform the Transaction Manager about top-level transactions are used for subtransactions. The only regard in which transaction processing is different is that subtransactions can be aborted without causing the parent transaction necessarily to abort. Subtransactions, however, may not be committed before their parents. Instead, when a parent requests to commit or abort, the servers acting on behalf of its subtransactions are treated as if they were part of the parent transaction.

3.7. Communication Manager

Like its counterpart in a standard Accent-based system, the Communication Manager is a single process that provides transparent node-to-node message forwarding. Location-transparent communication is achieved by interposing a pair of Communication Managers between the sender of a message and its intended recipient on a remote node. The Communication Manager supplies the sender with a local port to use for messages addressed to the remote process. Together with its counterpart at the remote node, the Communication Manager manages the network resource and implements the mapping between the local port used by the sender and the corresponding remote port belonging to the target process. Inter-node messages are delivered at most once and are not reordered.

In TABS, the Communication Manager also provides network monitoring. The Communication Manager is aware of the concept of transactions and is responsible for constructing the local portion of the spanning tree that the Transaction Manager uses during two-phase commit. In particular, the Communication Manager records the node's parent, if the transaction was initiated by a remote node, and the node's children, as defined in Section 3.6. It does this by monitoring request and response messages and using status information associated with each message. (This information is also used in detecting node crashes; see below.)

In conjunction with data servers and applications, the Communication Manager also detects permanent communication failures and aids in the detection of remote node crashes. There are three ways in which this is done in TABS:

- Loss of Connection. Loss of connection indicates the crash of a node with which this node is directly communicating. It is detected by the Communication Manager either because of the lack of a low-level acknowledgment or a negative acknowledgment indicating that the remote node no longer knows about the connection.

- Time-outs. Time-out, within an application or a data server, while waiting for a reply message catches errors such as infinite loops in remote data servers.

- Relationship Mismatch. This mechanism within the Communication Manager supports connectionless (datagram) communication between system components, such as the Transaction Manager, and is another way of detecting node crashes. In it, per-transaction information is used to check for inconsistencies in the relationship between sending and receiving nodes. This is done by including in every network message a description of the relation the sending node bears to the destination node; e.g., Parent, Child, NotRelated, etc. This information is used to update a per-transaction table that records what relation other nodes have to this one. When a node crashes, the Communication Manager's tables are lost. Therefore, following a crash, the Communication Manager will have no record of in-progress transactions, and messages from transactions that started prior to the crash will encounter a relationship mismatch.

4. How TABS Works

This section begins with a description of events that occur during the processing of a very simple transaction. The example does not demonstrate all of the facilities of TABS described in the previous section, but it does illustrate the most important interactions of TABS components. The latter part of this section describes how TABS handles events which are not part of normal transaction execution, such as crash recovery and system checkpoint.

4.1. A Simple Transaction

The example transaction uses two data servers and one application process as illustrated in Figure 4-1 and described as follows: The first data server, called the Array Server, implements an array of integers and is located on a separate node from that of the application process.

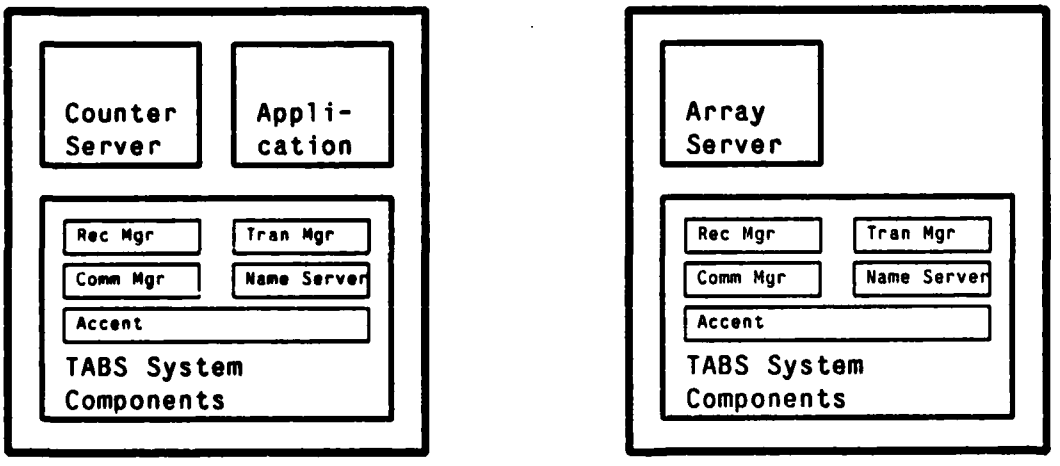


Figure 4-1: A System to Execute a Simple Transaction

This data server provides operations to read and write array elements, and sets read and write locks on individual elements. Value logging is used for recovery by this data server. The second data server, called the Counter Server, implements counters which may be incremented, decremented, and read. The Counter Server is located on the same node as the application process. The counters are implemented with type-specific locking so that different transactions may carry out multiple simultaneous increments and decrements. Because different transactions can concurrently update the same storage location, the Counter Server must use operation logging. The application process executes a transaction which is described by the pseudo-program in Figure 4-2. The transaction reads an array index and a value from a terminal and decrements a counter and an array element by the amount of the value.

4.1.1. Processing the Transaction

We assume that the application program keeps a cache with the ports and logical object identifiers for the array and counter objects. If this cache is not available, or if it is out of date, the application can request this information from the Name Server. After the ports and logical object identifiers are available, the application process initiates a transaction by sending a **Begin-Transaction** message to its local Transaction Manager and receiving a globally unique transaction identifier in a reply message.

Next, the application process reads an element of the array and then writes the array element and updates a counter. To read the array element, the application sends a message to the Array Server; in Figure 4-2 this is shown as a remote procedure call. After receiving the response, the application program sends requests to decrement the counter and write the array element in parallel.

The Array Server resides on a node different from that of the application process; therefore, the requests to that server are forwarded by the Communication Managers at each node. The Communication Manager at the application process' node notices when a new node joins the transaction and sends a **Has-Spread** message to the Transaction Manager informing it that the transaction has spread to another node.

When the Array Server receives the request to read an array element, it recognizes that this is the first operations is has done for the transaction; therefore, it sends a **Join** message to its local Transaction Manager to announce its participation in the transaction. This ensures that the data server will be notified when the transaction terminates. The Counter Server also notifies its local Transaction Manager when it first acts for the transaction.

```
VAR
  delta,                              {amount to decrement by}
  i,                                  {array index}
  number:  integer;                   {array value}

  cPort,                              {port to counter server}
  aPort:   Port;                      {port to array server}

  cLOI,                               {counter LOI}
  aLOI:    LOI;                       {array LOI}

  transID: TransactionID;
  result:  TransactionReturn;         {result of trying commit}

BEGIN
{ Get the ports, LOIs from the name server }
   { code not shown }

{ Get array index and amount by which to decrement }
   Read(i, delta);

{ Begin the transaction }
   transID := Begin-Transaction;

{ Fetch the array element }
   ArrayFetch(aPort, aLOI, transID, i, number);

{ Do procedure calls in parallel }
   IF (delta <= number) THEN COBEGIN
     ArrayStore(aPort, aLOI, transID, i, number-delta);
     Decrement(cPort, cLOI, TransID, delta);
   COEND
   ELSE Write('Delta too large');

{ Try to commit the transaction }
   result := End-Transaction(transID);
   IF (result = Success) THEN Write('Transaction committed!')
                         ELSE Write('Transaction aborted.');

END.
```

Figure 4-2: A Simple Transaction

While processing the requests to update the array element and counter, the data servers send messages containing **Modify-Records** to their local Recovery Managers for logging. Each **Modify-Record** identifies the modified object or portion thereof. The log record written by the Array Server contains the old and new values of the array element, while the log record written by the Counter Server contains an indication that the operation performed is a decrement and the amount of the decrement.

When the new value is written into the array element, the kernel will send a **Page-Modified** message to the local Recovery Manager since the page containing the array element is being modified for the first time[3]. The Recovery Manager logs a **Fetch-Record** for the page when it receives this message. It is the data server's responsibility to ensure that the **Page-Modified** message gets queued for the Recovery Manager *before* any messages containing **Modify-Records** referring to the page. The Array Server executes the following sequence of operations to ensure that messages get queued for the Recovery Manager in the proper order:

1. Obtain a write lock on the array element.
2. **Pin** the page containing the array element.
3. Store the new value into the array element. The kernel will enqueue a **Page-Modified** message to the Recovery Manager if the page is newly modified.
4. Send a **Modify-Record** to the Recovery Manager. Because the modifications have already been made, this message will arrive at the Recovery Manager after the message indicating the page was modified.
5. **Unpin** the page.

4.1.2. Transaction Commit

Commit processing begins when the application program sends the **End-Transaction** message to the local Transaction Manager, indicating that the transaction wants to commit. The local Transaction Manager then becomes the *coordinator* of the two-phase commit protocol.

Since the Transaction Manager knows that remote sites are involved, it first obtains the names of the children nodes from the Communication Manager; in this case, this is only the Array Server's node. It then forces a log record containing the name of this remote node and the name of the local Counter server in a **Collect-Record**, and sends **Prepare** messages to the Counter Server and to the Transaction Manager at the remote site.

In order to prepare the transaction, the Counter Server must do the following: refuse to accept any new operation requests for the transaction, ensure that the Decrement operation is finished, write **Modify-Records** for any changes that have not already been logged, and write a **Server-Prepared-Record** containing the transaction's locks. If the Counter Server is running and able to prepare, it will follow this procedure and reply affirmatively with a **Prepare-Acknowledgment**. Similarly, the remote Array Server will receive the forwarded **Prepare**

---

[3]Such a message might be sent when the counter is modified as well, however, in this example we assume that the page containing the counter is already dirty.

message, suspend the transaction, possibly write its log, and then reply affirmatively to its Transaction Manager. When the remote Transaction Manager realizes that the server on its site is prepared, it will force a **Prepare-Record** to the log and then return an affirmative reply to the coordinator.

When the coordinator sees that both the local server and the remote node have prepared, it knows that it can commit the transaction. It will force a **Commit-Record** to the log and then send **Commit** messages to the local server and the remote Transaction Manager. As soon as the coordinator logs this record, the transaction is committed. When the servers receive their commit messages, they drop the locks that they were holding on behalf of this transaction, and send **Commit-Acknowledgment** messages to their respective Transaction Managers. The remote node's transaction manager also return's a **Commit-Acknowledgment** message to the coordinator. When a transaction manager has received all appropriate messages, it at last writes a **Done-Record** to the log. This record indicates that the two-phase commit protocol has been completed for this transaction.

4.1.3. Transaction Abort

To illustrate how abort is implemented, assume that the Counter Server detects an underflow while processing the Decrement operation and sends a **Kill-Transaction** message to the local Transaction Manager. To initiate abort, the Transaction Manager sends a **Suspend-Transaction** message to the local Counter Server telling it to refuse new operation requests on behalf of this transaction, finish current operations, and log all its **Modify-Records**. Once this has been done, it is certain that the complete modify history is in the log for the Recovery Manager to examine.

Once the Counter Server has told the Transaction Manager that it has logged its changes, the Transaction Manager sends a **Roll-Back** message to the Recovery Manager instructing it to undo the changes that this transaction has made. To do this, the Recovery Manager finds the most recent **Modify-Record** of the aborting transaction. This record is sent to the Counter Server, and the server undoes the Decrement operation. The server then replies to the Recovery Manager, and--if necessary--the Recovery Manager repeats this process for each of the aborting transaction's **Modify-Records**. In this example, there is only one **Modify-Record** in the log.

When this process is finished, the Recovery Manager tells its Transaction Manager that roll-back is finished. Then, the Transaction Manager logs an **Abort-Record**, sends an **Abort** message to the Counter Server telling it to drop its locks, and informs the application program that the transaction is aborted. Once this is done, abort processing has been completed at this node. All that is left is to do is to notify the initiating application and the other server of the fact that the transaction is aborted. An **Abort** message is sent to the Array Server's node where the abort procedure is repeated.

4.2. Other Operations Processed by TABS

The operations discussed above are performed as a direct consequence of transaction processing; TABS also performs certain operations in response to system events initiated either by TABS processes or by the Accent kernel. The three operations discussed below are updating a disk page belonging to a recoverable segment, system checkpoint, and node restart.

4.2.1. Updating a Disk Page

When the Accent kernel wishes to copy to disk an unpinned page that is part of a recoverable segment, it sends an **About-to-Flush** message to the Recovery Manager indicating the page it desires to update. Data servers ensure that this message will arrive at the Recovery Manager after any messages containing **Modify-Records** by keeping modified pages pinned until after the messages containing **Modify-Records** are sent. The Recovery Manager replies to the **About-to-Flush** message from the kernel when the requirements of the write-ahead log protocol are satisfied. That is, the Recovery Manager replies when it has ensured that all log records pertaining to the page have been forced to disk. If recovery for some objects on the page is implemented using the operation logging algorithm, the log sequence number of the most recent log record referring to the page is included in the reply, and written by the kernel to non-volatile storage along with the page. Once the kernel has received the reply from the Recovery Manager, it is free to update the page. After completing the update, the kernel sends another message to the Recovery Manager to cause an **End-Write-Record** to be logged.

4.2.2. Checkpoint

Checkpoints serve to limit the amount of work performed during crash recovery. The Recovery Manager decides when to take a checkpoint, based on the amount of log written since the previous checkpoint. When a checkpoint is to be taken, the Recovery Manager sends a message to the Transaction Manager, advising it that transactions exceeding a maximum lifetime should be aborted. If long running transactions can be aborted at checkpoint time, it will not be necessary to undo their effects after a crash.

After long running transactions are aborted, the Recovery Manager sends a message to each local data server, requesting that it quiesce itself. The data servers reply once all of their objects are in an operation-consistent state and all their pages of recoverable segments are unpinned. The Recovery Manager collects the replies from data servers until all are quiescent and then proceeds to determine which pages of recoverable segments are dirty in volatile memory.

Because the kernel uses a least recently used page replacement algorithm, frequently accessed

pages of recoverable segments could remain in volatile indefinitely and during node restart it would be necessary to read indefinitely far back in the log to restore these pages. Therefore, after data servers are quiesced, the Recovery Manager sends a message to the kernel indicating old dirty pages that should have their disk representations updated as soon as possible. As usual, the kernel will ask the Recovery Manager for permission to copy each page, and inform the Recovery Manager after I/O is complete. Though it is important that dirty pages should not remain in volatile memory indefinitely, the TABS recovery algorithms do not require all modified pages of recoverable objects to be written to non-volatile storage during a checkpoint.

The Recovery Manager next sends another message to the Transaction Manager, requesting the list of the identifiers of incomplete and prepared transactions. The Recovery Manager constructs the **Checkpoint-Record** using this information and a list of dirty pages in volatile memory. The lists in the checkpoint record are used during crash recovery to determine when recovery is complete. After the Recovery Manager logs the **Checkpoint-Record**, it sends another round of messages to the data servers, informing them that they may resume normal processing.

4.2.3. Node Restart

Node failure recovery is coordinated by the Recovery Manager, which is driven by the data in the log. To carry out the recovery algorithms, the Recovery Manager must know the fate of the transactions mentioned in **Modify-Records**. However, the **Commit-Record**, **Abort-Record**, and other log records that contain transaction status information are constructed by the Transaction Manager, and the Recovery Manager interprets very little of the data it writes in the log on behalf of other TABS components. During node failure recovery, the Recovery Manager forwards these records to the Transaction Manager without examining them. When the Recovery Manager needs to know the status of a transaction in order to update its own data structures, it sends an inquiry to the Transaction Manager to obtain the information. It is the Transaction Manager's responsibility to inform Transaction Managers at other nodes about the transaction's status, as dictated by the two-phase commit protocol.

In maintaining the various data structures required by the recovery algorithms, the Recovery Manager interprets only those parts of **Modify-Records** that do not contain type-specific information. The Recovery Manager uses its data structures to determine whether the value of an object should be restored, or whether an operation on an object should be undone or redone. If so, it forwards the appropriate type-specific portion of the **Modify-Record** to the data server that wrote it. The data server then restores the object as required.

While node failure recovery is in progress, the failed node is unavailable for processing requests from data servers and applications at other nodes. After recovery is complete, the Recovery Manager sends a message to each data server, signalling that normal operation may be resumed.

5. Status and Evaluation

The TABS prototype described in this paper is an experimental system that is being built to evaluate our approach to constructing reliable distributed systems using transactions on abstract types. This section describes the status of the system and our plan for experimenting with the prototype. We conclude with preliminary evaluations of the design of the TABS prototype and our approach to distributed transactions.

5.1. Status of the TABS Prototype

At the time of this writing we have integrated the TABS components (modified Accent kernel, Communications Manager, Name Server, Recovery Manager, and Transaction Manager) and executed some simple transactions. The carefully constructed interfaces between these components permitted the components to be tested in isolation, and simplified the integration process.

At the time of this writing (July 1984), the procedure libraries that are used by programmers implementing applications and data servers are still being coded. Until this support is available, programming data servers will be a particularly difficult task. Certain features of the TABS Prototype discussed in this paper will not be available in the early version. The Recovery Manager and Accent kernel will support only the value-based recovery algorithm. Media failure recovery will not be implemented until some form of stable storage (e.g., dual disks) is available. And we will rely upon time-outs for deadlock detection.

5.2. Research Plans

Research into the uses of transactions in distributed systems will continue in two primary areas. The first area is experimenting with the TABS prototype to evaluate its design and to provide experience with the construction of atomic abstract types. The second area is the extension of the TABS system model and the design of better integrated operating system support for transactions on abstract types. Not surprisingly, work in this area will be influenced by results obtained from research in the first area.

5.2.1. Experimenting with the TABS Prototype

There are two types of experiments that will be conducted with the TABS prototype. First, the performance of the system for various sorts of simple local and distributed transactions will be measured. Second, various data servers and distributed applications will be constructed and evaluated.

Measurements of the performance of the TABS prototype must be designed to capture relevant data about the algorithms used by TABS, rather than inefficiencies of the prototype implementation. For this reason, measurements of the number of messages, amounts of log data written, number of log forces, and number of protection domain changes required for a transaction execution are more important than the amount of time and number of instructions required for transaction execution, or the working set size of the TABS system.

The applications that will be constructed on top of the TABS prototype range from a single simple data server to complete systems that use several sophisticated data servers. To gain early experience with programming a data server of moderate complexity, a B-Tree [Comer 79] package which stores all data in virtual memory has been written for Accent. This package will be modified to support the TABS recovery and synchronization models. Eventually, we will be able to compare the performance of B-Tree implementations which use value logging for recovery with the performance of implementations which use operation logging.

In order to gain experience with larger applications constructed using the TABS prototype facilities, we are planning to implement a distributed mail system with functionality similar to that of Grapevine [Birrell 82]. Unlike Grapevine, our mail system will store its registry database in a replicated directory that guarantees the consistency and availability of data despite failures of individual storage nodes [Daniels 83, Bloch 84]. The replicated directory algorithm is based on Gifford's weighted voting algorithm for files [Gifford 79]; like Gifford's algorithm it assumes the existence of an underlying distributed transaction facility. Our mail system will also replicate mail boxes, so that users may receive mail even if a node storing a replica of a user's mail box is unavailable.

5.2.2. Extending TABS

Extensions to TABS which we plan to consider in the near future include support for parallelism within individual data abstractions, support for the detection and resolution of deadlocks, and the further integration of TABS services with the operating system.

Parallelism will be realized within individual data abstractions by using *server classes*: groups of processes that share a common recoverable segment. Server classes represent a significant extension to the Accent model, which currently does not permit processes to communicate with each other via shared memory. Server classes, however, are not intended to encourage arbitrary communication through shared memory.

The TABS prototype as described in this paper does not provide any support for deadlock detection. Deadlock detection will be difficult in TABS because data servers maintain their own lock information. We envision that the subroutine libraries provided to data server implementors could contain interfaces which could be called by a deadlock detector. Some deadlock information can also be obtained from knowledge of which processes are waiting for messages on which ports.

From the standpoints of flexibility and modularity, the decomposition of TABS into modules has proven satisfactory. The weaknesses of the present implementation stem from the costs of communication among components. A future reorganization of the system could remedy this shortcoming by including more of the functions of the Recovery and Transaction Managers in the Accent Kernel. Other functions of these components and the Communications Manager could be combined in a single process.

5.3. Preliminary Evaluations

The TABS system described in this paper is a prototype, and its organization reflects this. The TABS components are largely autonomous, allowing considerable freedom for experimentation in their design. As long as the interface specifications remain unchanged, new implementations of TABS components can easily replace old ones. Isolating each component in an Accent process, with a private virtual address space and an explicitly defined message interface, encourages modularity and protects components from each other's errors.

However, the autonomy in TABS extracts a cost in performance. Forcing the TABS components to communicate by exchanging messages increases the cost of interactions that are potentially inexpensive. In addition, the isolation of components means that different components must often maintain redundant information. For example, the Recovery Manager keeps a list recording dirty pages of recoverable segments, even though the Accent kernel has similar information.

The TABS system model represents a significant compromise between the approaches to implementation of distributed transactions taken by database management systems [Gray 81b], and transactional file systems [Israel 78]. Database systems efficiently implement recovery and

synchronization for a specific application, but it is difficult to add new abstractions or to compose operations on separately implemented applications. Transactional file systems, on the other hand, provide a single low level abstraction upon which various composable applications may be built, but it is not possible to take advantage of type-specific optimizations in the implementation of synchronization and recovery. TABS assigns responsibility for type-specific aspects of synchronization and recovery to data servers, while providing common transaction management services and recovery facilities that can efficiently support multi-page objects implemented in pageable virtual memory. The TABS prototype provides a vehicle for experimentation with and evaluation of this new model for building distributed systems.

References

[Allchin 83a] James E. Allchin, Martin S. McKendry.
Facilities for Supporting Atomicity in Operating Systems.
Technical Report GIT-CS-83/1, Georgia Institute of Technology, January, 1983.

[Allchin 83b] J. E. Allchin, M.S. McKendry.
Synchronization and Recovery of Actions.
In *Proc. of the Second Annual ACM Symp. on Principles of Distributed Computing*, pages 31-44. August, 1983.

[Birrell 82] Andrew D. Birrell, Roy Levin, Roger M. Needham, and Michael D. Schroeder.
Grapevine: An Exercise in Distributed Computing.
Comm. of the ACM 25(4), April, 1982.

[Bloch 84] Joshua J. Bloch, Dean S Daniels, Alfred Z. Spector.
Weighted Voting for Directories: A Comprehensive Study.
Carnegie-Mellon Report CMU-CS-84-114, Carnegie-Mellon University, Pittsburgh, PA, April, 1984.

[Comer 79] Douglas Comer.
The Ubiquitous B-Tree.
ACM Computing Surveys 11(2):121-137, June, 1979.

[Daniels 83] Dean Daniels, Alfred Z. Spector.
An Algorithm for Replicated Directories.
In *Proc. of the Second Annual ACM Symp. on Principles of Distributed Computing*, pages 84-113. August, 1983.

[Gifford 79] David K. Gifford .
Weighted Voting for Replicated Data.
In *Proc. Seventh Symp. on Operating System Principles*, pages 150-162. ACM, 1979.

[Gray 78] James N. Gray.
Notes on Database Operating Systems.
In R. Bayer, R. M. Graham, and G. Seegmuller (editors), *Operating Systems - An Advanced Course*, pages 393-481. Springer-Verlag, 1978.
Also available as IBM Research Report RJ2188, IBM San Jose Research Laboratories, 1978.

[Gray 80] James N. Gray.
A Transaction Model.
IBM Research Report RJ2895, IBM Research Laboratory, San Jose, CA, August, 1980.

[Gray 81a] James N. Gray.
The Transaction Concept: Virtues and Limitations.
In *Proc. of Very Large Database Conference*, pages 144-154. September, 1981.

[Gray 81b] James N. Gray, et al.
The Recovery Manager of the System R Database Manager.
ACM Computing Surveys (2):223-242, June, 1981.

[Israel 78] Jay E. Israel, James G. Mitchell, Howard E. Sturgis.
Separating Data from Function in a Distributed File System.
In *Proc. 2nd Int'l Symp. on Operating Systems*. IRIA, 1978.

[Korth 83] Henry F. Korth.
Locking Primitives in a Database System.
Journal of the ACM 30(1), January, 1983.

[Lampson 81] Butler W. Lampson.
Atomic Transactions.
In G. Goos and J. Hartmanis (editors), *Distributed Systems - Architecture and Implementation: An Advanced Course*, chapter 11, pages 246-265. Springer-Verlag, 1981.

[Lindsay 79] Bruce G. Lindsay, et al.
Notes on Distributed Databases.
IBM Research Report RJ2571, IBM Research Laboratory, San Jose, CA, July, 1979.

[Liskov 82] Barbara Liskov and Robert Scheifler.
Guardians and Actions: Linguistic Support for Robust, Distributed Programs.
In *Proceedings of the Ninth ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 7-19. Albuquerque, NM, January, 1982.

[Liskov 83] B. Liskov, M. Herlihy, P. Johnson, G. Leavent, R. Scheifler, W. Weihl.
Preliminary Argus Reference Manual.
Programming Methodology Group Memo 39, Massachusetts Institute of Technology, Laboratory for Computer Science, October, 1983.

[Lorie 77] Raymond A. Lorie.
Physical Integrity in a Large Segmented Database.
ACM Trans. on Database Systems 2(1):91-104, March, 1977.

[Mohan 83] C. Mohan, B. Lindsay.
Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions.
In *Proc. of the Second Annual ACM Symp. on Principles of Distributed Computing*, pages 76-88. August, 1983.

[Moss 81] J. Eliot B. Moss.
Nested Transactions: An Approach to Reliable Distributed Computing.
PhD thesis, MIT, April, 1981.

[Rashid 81] Richard Rashid, George Robertson.
Accent: A Communication Oriented Network Operating System Kernel.
In *Proc. Eighth Symp. on Operating System Principles*. ACM, 1981.

[Rashid 83] Richard F. Rashid.
Accent Kernel Interface Manual.
November, 1983.

[Reed 78] David P. Reed.
Naming and Synchronization in a Decentralized Computer System.
PhD thesis, MIT, September, 1978.

[Schwarz 83] Peter M. Schwarz, Alfred Z. Spector.
Recovery of Shared Abstract Types.
Carnegie-Mellon Report CMU-CS-83-151, Carnegie-Mellon University, Pittsburgh, PA, October, 1983.
Forthcoming.

[Schwarz 84a] Peter M. Schwarz, Alfred Z. Spector.
Synchronizing Shared Abstract Types.
Transactions on Computer Systems 2(3), July, 1984.
Also available as Carnegie-Mellon Report CMU-CS-83-163, November 1983.

[Schwarz 84b] Peter M. Schwarz.
Transactions on Typed Objects.
PhD thesis, Carnegie-Mellon University, July, 1984.
Forthcoming.

[Spector 83] Alfred Z. Spector, Peter M. Schwarz.
Transactions: A Construct for Reliable Distributed Computing.
Operating Systems Review 17(2):18-35, April, 1983.
Also available as Carnegie-Mellon Report CMU-CS-82-143, January 1983.

[Traiger 82] Irving L. Traiger.
Virtual Memory Management for Database Systems.
IBM Research Report RJ3489, IBM Research Laboratory, San Jose, CA, May, 1982.

[Weihl 83] William E. Weihl.
Data Dependent Concurrency Control and Recovery.
In *Proc. of the Second Annual ACM Symp. on Principles of Distributed Computing*, pages 73-74. August, 1983.

END

FILMED

12-84

DTIC